Assignment 1
Word Completion Report

COSC 2123/1285 Algorithms and Analysis

Student A: Jabbar Baloghlan [s3890406]

Student B: Sisi Zhang [s4000130]

1. Data Generation and Timing Setup

This section explains the setup and code used to test and generate experimental data for the project.

1.1 Data File Generation

To get randomized datasets for our program and testing, we created the ‘data_gen.py’ program. The program uses the ‘sampleData200k’ dataset to get the possible entries for future datasets. The program gets a desired data size and creates the datasets with random, non-repetitive entries from the ‘sampleData200k’ dataset. The output file is named ‘dataN.txt’ where N is the imputed size of the new array.

1.2 Test File Generation

In addition to the randomized dataset generation, ‘data_gen.py’ also generates an ‘testN.in’, where N is the imputed size of the new array, an instructions file to test for correctness of the code, dataset execution, and provide way to get the timings for each of the functions in the code (see section 1.4). The instructions file contains 4 of each of the ADD, SEARCH, DELETE, AND AUTOCOMPLETE, instructions to test. Each half of the instructions also have tests for entries for words that are and are not in the dataset (the instructions for existing words are dynamically generated together during dataset generation to avoid issues). Additionally, the code generated instructions for both short and long words (see section 1.3)

1.3 Consideration of Word Length

Each of the inverse halves of the instructions use short words (less than 5 letters) and long words (more than 5 letters). This was designed to get a broader spectrum of results and timings for the analysis of the project. The AUTOCOMPLETE instruction also gets both the short and the long words. Reducing the obtained words by 3 letters, we finally could test the feature on both ends of word length.

1.4 Runtime Measurement

To measure the runtime of each of the functions of the code, and the code itself, we modified the test scripts. This was done to enable accessibility of executing any dictionary type, and getting the same margins of time, evading individual differences in timings (each directory could behave differently to measurements). The time code we decided to use is the ‘timeit’ library, which provides the most precise runtime than other libraries that we tested (ex. ‘time’ library). The modifications are as follows:

- ‘dictionary_test_script.py’: This script contains generalized timing code to acquire the overall runtime of our program
- ‘dictionary_file_based.py’: In this script, we added timing code into each functions, allowing us to record the execution times specific to each of the functions. After each capture of execution times, the result is outputted into the terminal, where it can be easily gathered.

1.5 Testing setup

For our testing, we decided to use 6 different sizes of datasets of: 500, 1000, 5000, 10000, 50000, and 100000. First, we generated the datasets and instruction files of given sizes. Then, we used the test scripts to test the code and gather the timings necessary for our analysis. After execution for each of the datasets and dictionary types, we gathered the runtimes from the terminal, and checked the ‘.out’ files for any abnormalities in behavior during execution. Ensuring successful runs, we proceeded to analyze the results

2. Evaluation and Analysis of Experimental Results

In this section, it presents a comprehensive analysis of the outcomes derived from the team's experimentation with various data structures, including Array, Linked List, and Trie. These experiments span a range of parameter settings and data sizes, all implemented by student A. Our analysis focuses on four fundamental operations: Add (A), Search (S), Delete (D), and Autocomplete (AC). To facilitate this analysis, it relies on the computation of average running times, a key metric employed to construct graphical representations for visual comparisons between observed performance and theoretical time complexities.

2.1 Array

Average Running Times

The table below showcases the average running times for each operation within the Array data structure across varying data sizes:

Operation /Data Size	*Add (A)*	*Delete (D)*	*Search (S)*	*Autocomplete (AC*
Data 500	0.0000183000	0.0000181000	0.0000100500	0.0000355500
Data 1000	0.0000467750	0.0000325750	0.0000231500	0.0000860000
Data 5000	0.0002154000	0.0001600000	0.0001020500	0.0003712500
Data 10000	0.0003464750	0.0003102750	0.0002168250	0.0006453250
Data 50000	0.0017588500	0.0016586000	0.0010230750	0.0033517250
Data 100000	0.0038110250	0.0033392250	0.0021801500	0.0068422000

In the case of Array, the dataset size does not exhibit uniform growth rates, necessitating further data manipulation to ensure uniformity in running times

Data Size	2	10	20	100	200
Growth Rate of Data Size vs. Growth Rate of Average Running Time					
Add (A)	2.556011374	11.77049554	18.93306594	96.11205376	208.2528018
Delete (D)	1.799723755	8.839783345	17.14227196	91.63539799	184.4876478
Search (S)	2.303481888	10.15422273	21.57461944	101.7984698	216.9302667
Autocomplete (AC)	2.419128792	10.4430406	18.15260438	94.2820133	192.4669798

Graphs

[Appendix I:Graphs 1]

Observing the individual graphs ***[See Figures 2.1.1-2.1.4]***, It notes a consistent upward trend in running times with increasing data size. However, the magnitude of the increase varies, with Add and Delete showing minimal fluctuations, while Autocomplete experiences the most significant impact from data size variations, reaching peaks of

0.007s. In contrast, Search execution remains within the range of 0.0025s. Detailed differences become apparent in ***section 2.1.5***, revealing that although growth trends are similar, coefficients differ. Notably, Autocomplete exhibits the highest coefficient among the operations, suggesting a more substantial sensitivity to data size.

Comparison with Theoretical Time Complexities

[See Figure 2.1.6]
The scalability plot serves to describe how algorithm or data structure performance scales with increasing dataset size, featuring the rate of average running time on the y-axis and the rate of data size on the x-axis. It encompasses all operations (search, add, delete, autocomplete) for the array-based data structure.

Simlarities	
Add	O(n) on average -This operation entails finding the correct position for insertion, necessitating element shifting. This is a linear operation on averagesince, on average, it may need to move half of the elements.
Delete	O(n) on average - Similar to insertion, deletion requires element shifting, making it a linear operation on average.
Differences	
Search	O(log n) - Binary search can be used on the sorted array to find a word, resulting in logarithmic time complexity. However, the actual coding simply uses for loops so that the result is O(n see ***Section 2.1.6***.
Autocomplete	O(p * log n) on average - This operation involves searching for prefix matches and sorting. It can perform a binary search to find the starting point of the words with that prefix(p) and then retrieve the top three words with the highest frequencies, which takes O(3 * log n) on average. In the actual coding,this runtime was tested based on the coding of Student A, who created 3 lists, where contain list was used to contain all items beginning with the prefix, and then sorted them into sorted_array, and best_array was used for the 3 most frequently occurring items, as opposed to Student B, who first created a duplicate list to be sorted, and then returned the top 3 relevant words, with a possible average complexit of O(3 * log n). see ***Appendix II***

2.2 Linked List

Average Running Times

This section presents the average running times for each operation within the Linked List data structure across various data sizes:

Operation /Data Size	*Add (A)*	*Delete (D)*	*Search (S)*	*Autocomplete (AC)*
Data 500	0.0000315750	0.0000256250	0.0000216250	0.0000536750
Data 1000	0.0000554000	0.0000446250	0.0000442500	0.0001032000
Data 5000	0.0003217000	0.0002625500	0.0002463250	0.0005286750

Data 10000	0.0005705500	0.0004848000	0.0004390250	0.0010005000
Data 50000	0.0031101500	0.0025854000	0.0022717000	0.0055874000
Data 100000	0.0063684000	0.0052867750	0.0053564000	0.0107552250

Also,the further comparison shown in the below is made with the Data 500 set for performance analysis and optimization, ensuring uniformity in running times.

Data Size	2	10	20	100	200
Growth Rate of Data Size vs. Growth Rate of Average Running Time					
Add (A)	1.754551764	10.18843746	18.06967128	98.50037238	201.6911626
Delete (D)	1.741464199	10.24585788	18.91903056	100.8936929	206.3132397
Search (S)	2.046243148	11.39075595	20.30174394	105.0497598	247.6949123
Autocomplete (AC)	1.922682769	9.849557465	18.63996272	104.0968768	200.3768001

Graphs

[Appendix I:Graphs 2]

It is similar to the Array, the Linked List exhibits an overall upward trend in running times with increasing data size. While Autocomplete is highly sensitive to data size, reaching up to 0.012s, and the other three operations (Add, Delete, Search) peak at around 0.006s.

Significant differences among these operations are detailed in ***section 2.2.5***, demonstrating that Autocomplete exhibits the highest coefficient, likely due to the impact of the given string as a prefix. It is important to note that, unlike the array-based dictionary, words in the linked list are not sorted, resulting in a wider range of execution times.

Comparison with Theoretical Time Complexities

Simlarities	
Search	Average Case - O(n). Searching an unsorted singly linked list requires traversing the list from the head node to the end,checking each node for a match. Which resulted in an average-case linear time complexity.
Delete	Average Case - O(n).It also involves traversing the list, leading to an average-case linear time complexity. Although O(1) is possible in the best-case scenario, on average, this may require traversing half of the list, resulting in O(n) time complexity.
Differences	
Add	O(1) on average - In the average case, adding a new word to the front of th linked list is a constant-time operation. It can be implemented simply to create a new node and update the head reference.

	However, the insertion is to add the element at the end in the actual progra which leads to the observed actual complexity is O(n),see ***section 2.2.6***.
Autocomplete	Considering the above factors, the average-case time complexity for autocomplete in a linked-list-based dictionary is generally O(n + m * log m), where n is the number of words in the list, and m is the number of words th match the given prefix. It consists of two components： 1. O(n): This operation requires iterating through the entire list and checkin each word to see if it matches the prefix. 2.O(m * log m): After finding all matching words, it would need to sort them by frequency. However in the actual case,there are very few words that match the prefix (is small) and is almost the same k in different sizes of data(in data_gen.py) so the time complexity may be closer to O(n).

2.3 Trie

Average Running Times

In contrast to the previous data structures, the performance of Trie-based dictionary is less influenced by data size, as reflected in the following data:

Operation /Data Size	*Add (A)*	*Delete (D)*	*Search (S)*	*Autocomplete (AC)*
Data 500	0.0000049000	0.0000022250	0.0000014000	0.0000050500
Data 1000	0.0000067250	0.0000027500	0.0000015500	0.0000067500
Data 5000	0.0000059250	0.0000026750	0.0000020250	0.0000129000
Data 10000	0.0000055750	0.0000027000	0.0000018750	0.0000078250
Data 50000	0.0000050500	0.0000037250	0.0000023500	0.0001999500
Data 100000	0.0000051250	0.0000033750	0.0000029000	0.0005081500

The table above showcases minimal fluctuations in running times across different data sizes, primarily influenced by the length of words rather than the dataset size.

Graph

[Appendix I:Graphs 3]

Despite a limited range of dataset in the experiments, it is still evident that longer words generally require more time to execute.***[Observation of Left Data]***

Consequently, it is easy to compare the growth rates of average lengths for the same operations to the Theoretical Time Complexities, which are the same growth rates of log average word lengths.***[Observation of Right Data]***

Comparison with Theoretical Time Complexities

Based on the calculation, it is almost certain that the actual running time is nearly identical to Theoretical Time Complexities.

Search	O(log l) on average, where l is the average length of the keys in the trie. When searching for a word, it traverses the trie character by character, making a decision at each level based on the character. In a balanced trie, this results in a logarithmic time complexity with respect to the length of the word.
Add/Delete	Similar to the search operation, it needs to find the location to insert/delete the word in the trie by traversing down its characters, and the complexity is logarithmic in the length of the word.
Autocomplete	In a well-balanced Trie, the number of nodes (or words) that match the prefix is roughly proportional to the depth of the Trie nodes representing the prefix, which is O(log p).Once the Trie nodes matching the prefix, it need to retrieve the word suggestions associated with these nodes. Hence,O(log(p + l)) is the average complexity, where p is the number of words with the given prefix, and l is the total number of characters in those words.

2.4 Overall Analysis

In general, Trie outperforms both Array and Linked List in terms of average execution times, as evidenced by the significantly smaller decimal values. Therefore, among these three data structures, Trie emerges as the most suitable choice for the overall project,i.e the Dictionary.

Excluding Trie, when comparing Array and Linked List, Array exhibits lower average execution times and growth rates. However, theoretical analysis indicates that Linked List holds an advantage in insertion (O(1)) and may outperform Array for autocomplete operations if the dataset is sufficiently small (n < m, m < 3) due to its O(n + m * log m) complexity, compared to Array's O(3 * log n).

If Trie is not a viable option and optimization is sought, consider the following suggestions:

1. Modify Linked List insertion to insert a new element at the head which can reduce the current time complexity to a constant.
2. To improve the efficiency of operations like SEARCH and AUTOCOMPLETE, consider sorting the dataset based on word frequencies. This would allow for binary search, reducing the time complexity to O(log n) for these operations..
3. Employ a copy list for sorting and return the top 3 related words for Autocomplete ***see Appendix I***.

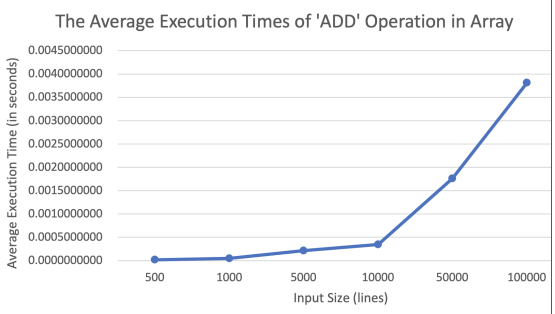
Additionally, For large datasets and frequent search and autocomplete operations, consider using more efficient data structures like binary search trees or hash tables, which offer faster retrieval times.

3. Appendix

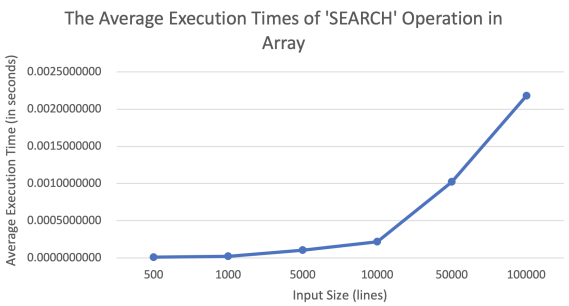
Appendix I:Graphs

1.Array-based

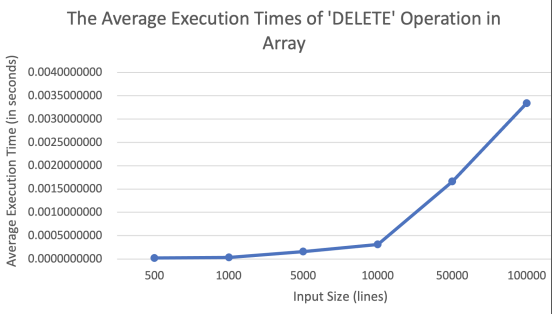
[Figure 2.1.1- Insert Graph]



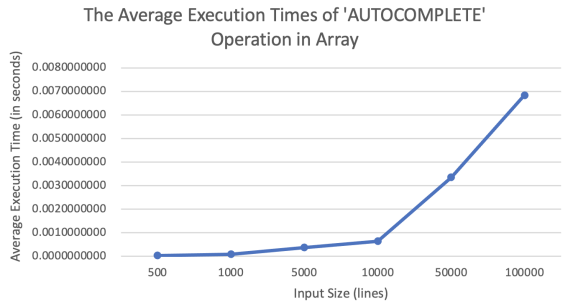
[Figure 2.1.2- Search Graph]



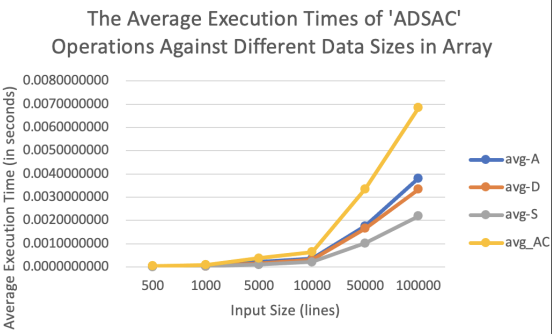
[Figure 2.1.3-Delete Graph]



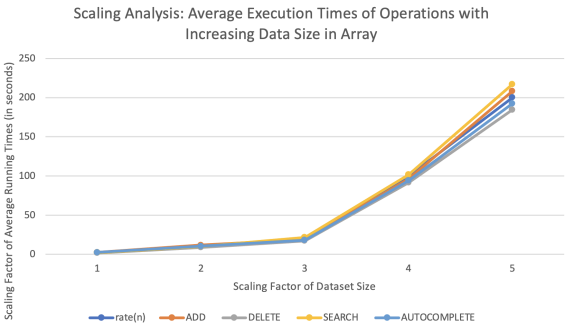
[Figure 2.1.4- Autocomplete Graph]



[Figure 2.1.5-Combination Analysis]



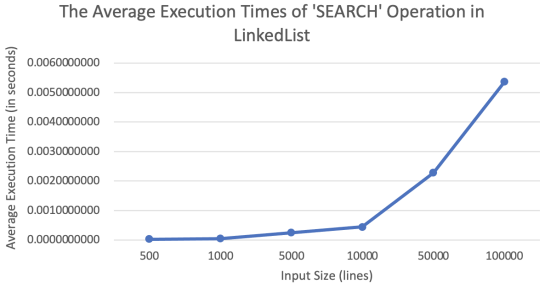
[Figure 2.1.6-Scaling Analysis]



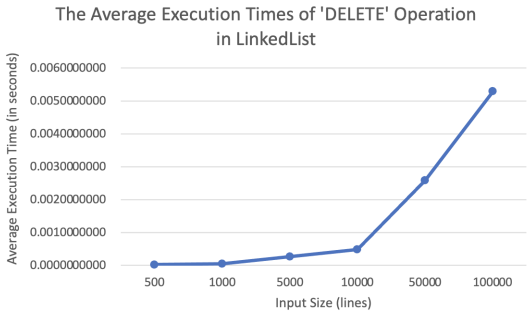
2.LinkedList_based

[Figure 2.2.1- Insert Graph]

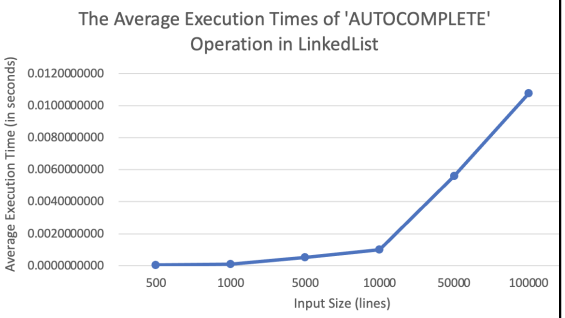
[Figure 2.2.2- Search Graph]



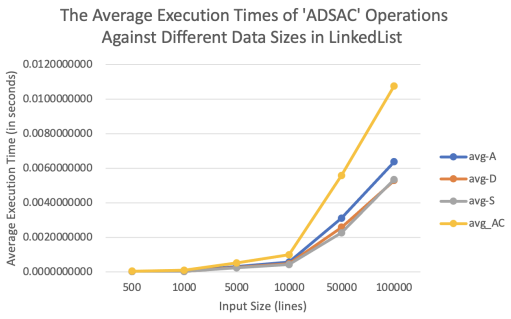
[Figure 2.2.3-Delete Graph]



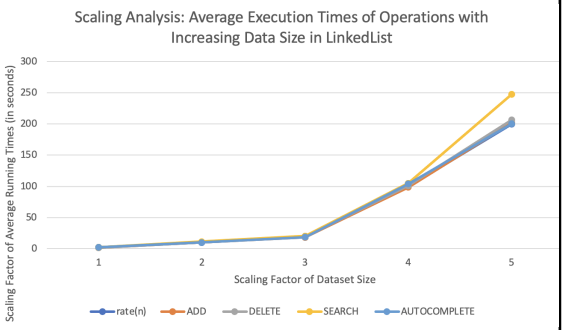
[Figure 2.2.4- Autocomplete Graph]



[Figure 2.2.5-Combination Analysis]



[Figure 2.2.6-Scaling Analysis]



3.Trie_based

avg short A	0.0000042333	rate for art	2.7637715700
avg Long A	0.0000117000	rate(logl)	2.5789019232
avg short D	0.0000027000	rate for art	1.0648076287
avg long D	0.0000028750	rate(logl)	1.6609640474
avg short S	0.0000016833	rate for art	1.178215967
avg long S	0.0000019833	rate(logl)	1.430676558
avg short AC	0.0000025500	rate for art	2.006530266
avg long AC	0.0000051167	rate(logK+l)	2.024369199

Appendix II:Supplementary documents

1.Code files

This appendix includes supplementary code files and scripts created by Sisi Zhang(Student B) and Jabbar Basloghlan (Student A). They provide additional context and implementation details for the array, linked list, and trie based data structures discussed in the report.

2.Analysis Excel File (A1_Analysis.xlsx)

This detailed Excel file contains all the data presented in the report, including raw experimental results, calculations, and any other data-related information. It serves as a comprehensive resource for anyone interested in a more in-depth analysis of the experimental data.

Appendix III: External resource

1.Sublime Text and Visual Studio Code

Sublime Text and Visual Studio Code are the code editors that were used for writing and editing code during the project. They are used to convert the running time to valuable data in this assignment.

2.Excel

Microsoft Excel was utilized for calculating observed data, performing data analysis, and creating the plots and tables presented in the report.